Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1995

Neural and Neuro-Fuzzy Approaches to Support ``Intelligent'' Scientific Problem Solving

Anupam Joshi

Sanjiva Weerawarana

Narendran Ramakrishnan

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
95-039

Joshi, Anupam; Weerawarana, Sanjiva; Ramakrishnan, Narendran; Houstis, Elias N.; and Rice, John R., "Neural and Neuro-Fuzzy Approaches to Support ``Intelligent'' Scientific Problem Solving" (1995). *Department of Computer Science Technical Reports.* Paper 1215.
https://docs.lib.purdue.edu/cstech/1215

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Neural and Neuro-Fuzzy Approaches to Support "Intelligent" Scientific Problem Solving

Anupam Joshi
Sanjiva Weerawarana
Narendran Ramakrishnan
Elias N. Houst
John R. Rice
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-95-039
June, 1995

Neural and Neuro-Fuzzy Approaches to Support "Intelligent" Scientific Problem Solving *

Anupam Joshi, Sanjiva Weerawarana, Narendran Ramakrishnan, Elias N. Houstis, John R. Rice
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA.
Phone: (317)-494-7821, Fax: (317)-494-0739
Email: {joshi,saw,ramakris,enh,jrr}@cs.purdue.edu

June 14, 1995

Abstract

Scientific computing uses computers, especially High Performance Computing (HPC) systems, to solve complex mathematical equations which model physical phenomena. Using these systems now requires expert knowledge in a variety of fields of computer science, such as parallel computing and numerical methods. This often makes application scientists, who have the domain expertise to devise the mathematical models, unable to use the power of HPC systems. The object of problem solving environments (PSEs) is to create software systems that hide the details and complexity of the system from the users, and to allow them to deal with a high level, abstract entity that understands the application domain "language". This requires approximate reasoning techniques to automate much of numerical and parallel computing, as well as to interpret the users input. Over the past several years, we have developed PYTHIA, an "intelligent" computational assistant to achieve this goal. In this paper, we describe the connectionist techniques used in developing PYTHIA. Specifically, we discuss backpropagation based systems, as well as hybrid neuro-fuzzy systems which we have developed and used. We also compare the performance of these alternative approaches with each other, as well as with naive classifiers.

1 Introduction

The scientific process has traditionally had two components, theoretical and experimental. Computation has now become the third component of the scientific process. It allows scientists to devise mathematical models of the physical phenomenon, and then simulate them computationally. One of the major factors in the development of the computational component has been the advent of HPC to handle compute intensive applications. In fact, many hitherto dormant and difficult challenges in applied sciences, such as modelling protein folding or internal combustion engine design, have become feasible to attack using HPC power. The capabilites provided by HPC power have made scientific computing a rapidly growing field.

Yet, for all its potential payoffs, the state–of–the–art in HPC does not match that of its workstation/PC cousin in terms of ease of use. In fact, Diane O'Leary in a recent article[8] went so far as to compare parallel computing of today to the "prehistory" of computing, where computers were used by a select few who understood the details of the architecture and operating system, where programming was complex, and debugging required reading hexadecimal dumps. Computer time had to be reserved, jobs were submitted in batches, and crashes were common. Users were never sure of whether an error was due to a bug in their code or in the system. Most users of HPC find themselves in this situation. Clearly, if the HPC based computational paradigm for the scientific process is to succeed and become ubiquitous, it must provide the simplicity of access that became popular with the advent of point and click capability of PCs.

*This work was supported in part by NSF awards ASC 9404859 and CCR 9202536, AFOSR award F49620-92-J-0069 and ARPA ARO award DAAH04-94-G-0010

An important recent advance in this direction is the development of Problem Solving Environments(PSEs). Even in the early 1960s scientists began to envision problem-solving computing environments not only powerful enough to solve complex problems, but also able to interact with users on human terms. The rationale of our research in this area is that the dream of the 1960s can be the reality of the 1990s, high performance computers combined with better algorithms and better understanding of computational science have put PSEs well within our reach.

A Problem Solving Environment (PSE) is a computer system that provides all the computational facilities necessary to solve a target class of problems[4]. These features include advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems and provide a "natural" interface, so users can use them without specialized knowledge of the underlying computer hardware or software. By exploiting modern technologies such as interactive color graphics, powerful processors, and networks of specialized services, PSEs can track extended problem-solving tasks and allow users to review them easily. Overall, they create a framework that is all things to all people: they solve simple or complex problems, support rapid prototyping or detailed analysis; they can be used in introductory education or at the frontiers of science. In order to develop systems that are truly easy to use, PSEs need to provide the user with a high level abstraction of the complexity of the underlying computational facilities. The user can not, and should not, be expected to be well versed in selecting appropriate numerical, symbolic and parallel systems, along with their associated parameters, that are needed to solve a problem.

An important task of a PSE is to accept some "high level" description of the problem from the user, and then automatically select the appropriate computational resources (hardware, software) needed to solve the problem. Clearly, this task requires the use of "intelligent" techniques – it requires knowledge about the problem domain and reasoning strategies. In this paper we report on PYTHIA, an intelligent assistant that serves as a part of various PSEs and uses several "soft" computing techniques to achieve this goal. We begin by describing the specific problem that PYTHIA addresses. We then report results from the various connectionist reasoning strategies we tested on this system, and show how they compare with each other. We describe how we are using a neuro-fuzzy technique we have developed to further enhance PYTHIA by allowing it to operate in an multiagent environment.

2 PYTHIA

PYTHIA attempts to solve the problem of determining an optimal strategy (i.e., a solution method and its parameters) for solving a given problem within user specified resource (i.e., limits on execution time and memory usage) and accuracy requirements (i.e., level of error). While the techniques involved are general, our current implementation of PYTHIA operates in conjuction with systems which solve (elliptic) partial differential equations (//ELLPACK [5]). In the rest of this paper, whenever we refer to a "problem" in the context of implementation and testing, we mean a PDE problem.

PYTHIA accepts as input the description of a problem, and produces the method(s) appropriate to solve it. Its strategy is similar to that believed to underlie human problem solving skills. There is a wealth of evidence from psychology that suggests that humans compare new problems to ones they have seen before, using some metric of similarity to make that judgement. They use the experience gained in solving "similar" previous problems to evolve a strategy to solve the present one. This same strategy has been defined as "case based" reasoning in the AI literature. Similar transformational strategies also formed the basis of some early AI problems solvers, such as GPS and SAINT, which seek to reduce problems into smaller ones, or to those seen before. In effect, the strategy of PYTHIA is to compare a given problem to the ones it has seen before, and then use its knowledge about the performance characteristics of prior problems to estimate those of the given one.

Thus, to recommend a strategy to solve a given PDE problem, p, PYTHIA needs:

- a database P of previously solved problems along with data on the effectiveness of various solution methods on those problems,
- a mechanism to identify the problems from the database that are similar to p and,
- comparative data on the effectiveness of various methods on the problems in the database.

Operator	Boundary Conditions	PDE Functions	Solution
Poisson	Dirichlet	Smooth	Singular
Laplace	Neumann	Oscillatory	Analytic
Helmholtz	Mixed	Wave Front	Oscillatory
Self-adjoint	Homogeneous	Singular	
Homogeneous		Peak	

Table 1: Examples of PDE problem characteristics for elliptic partial differential equations.

With this information, PYTHIA uses the following algorithm to select the method to be used :

1. Analyze the PDE problem and identify its characteristics. This stage involves applying symbolic analysis to extract some characteristics and asking the user about characteristics that cannot be determined automatically.

2. Identify the problem $q \in P$, whose characteristics most closely match that of the new problem p.

3. Use the performance data for q to predict the method to use for p and the values for appropriate parameters to achieve the specified computational and performance objectives.

Clearly, as the size of P increases, comparing the new problem to all those in P becomes increasingly time intensive. An alternate strategy we use splits the previously seen problems into classes. Rather than search all the previous problems to find the most similar one, we confine the search to the ones that are in the same class as the new problem. Thus the strategy becomes:

1. Analyze the PDE problem and identify its characteristics. This stage involves applying symbolic analysis to extract some characteristics and asking the user about characteristics that cannot be determined automatically.

2. Identify the set $S \subset P$, where S is the subset of problems whose characteristics are similar to those of p.

3. Identify the problem $q \in S$, whose characteristics most closely match those of p.

4. Analyze the performance data for the problems in S and rank the applicable methods to select the "best" method for solving the problem within the given computational and performance objectives. Use the performance data available for q to predict the values for appropriate parameters to achieve the specified computational and performance objectives.

PYTHIA consists of several components: the database P of previously seen problems and their related performance information, a set of problem classes, a knowledge base of performance rules for interesting classes of problems, and, of course, the inferencing environment that analyzes with these components.

2.1 PDE Problem Characteristics and Their Extraction

In this section we identify some of the characteristics of a PDE problem that are important for PYTHIA to detect correlations between the values of a certain characteristic and the suitability or effectiveness of a particular solution method. We assume that the PDE problem is defined in terms of the following components: The PDE operator and right hand side, the initial and boundary conditions, and the spatial and time domains of definition.

There are two main types of characteristics of a PDE problem: Characteristics of the problem components and characteristics of the solution. The characteristics of PDE problem components include some classification information (for example, whether the operator is homogeneous or not) and some quantitative information about the behavior (for example, smoothness and local variation) of the PDE functions (i.e., coefficients of the operators, right hand side of the operators, boundary and initial conditions, and the solution).

Table 1 shows some of the characteristics that we use to characterize a PDE problem and its solution. Each characteristic is also associated with a value α, where $\alpha \epsilon [0, 1]$. ($\alpha = 0$ means pure absence of that property while $\alpha = 1$ means pure presence.) For logical characteristics (for example, whether the boundary conditions

PROBLEM #28	$(w\,u_x)_x + (w\,u_y)_y = 1,$ where $w = \begin{cases} \alpha, & \text{if } 0 \le x, y \le 1 \\ 1, & \text{otherwise.} \end{cases}$
DOMAIN	$[-1,1] \times [-1,1]$
BC	$u = 0$
TRUE	unknown
OPERATOR	Self–adjoint, discontinuous coefficients
RIGHT SIDE	Constant
BOUNDARY CONDITIONS	Dirichlet, homogeneous
SOLUTION	Approximate solutions given for $\alpha = 1, 10, 100$. Strong wave fronts for $\alpha \gg 1$.
PARAMETER	α adjusts size of discontinuity in operator coefficients which introduces large, sharp jumps in solution.

Figure 1: A problem from the PDE population.

are Dirichlet or not), we use the values 0 and 1 for false and true, respectively. The set of characteristics of a PDE problem are represented as a *characteristic vector* v, which PYTHIA uses to identify a similar PDE problem or a class of related PDE problems from P. The vector

$$\left(\underbrace{2}_{1}, \underbrace{0,0,0,1,0,0,0,0,1,0,1,0,0}_{2}, \underbrace{0.0,0.0}_{3} \right)$$

represents the type characteristics of the PDE operator of the problem specified in Figure 1. The first number indicates that the operator is two dimensional. The second set of numbers indicate that the operator is not Poisson, not Laplace, not Helmholtz, is self–adjoint, does not have constant coefficients, does not have single derivatives, does not have mixed derivatives, is not homogeneous, is linear, is not nonlinear, is elliptic, is not parabolic and is not hyperbolic, respectively. The last two numbers are the subjective measures of the smoothness and local variation properties of the operator. In this study, we have assumed only linear PDE problems and solvers.

2.2 Problem Database and Class Definitions

The classes in P can be obtained in two ways. First, one can apply one of several clustering mechanisms to determine the clusters in the database, and set each of these to be a class. Second, domain experts can define some classes *a priori* based on the characteristics of the problem at hand. For some simplistic cases, one can precisely define a mapping of characteristics of the problem into a class. For most classes however, such mappings cannot be defined, and one of the tasks before PYTHIA is to learn these mappings. We have used several techniques, traditional as well as neural, to this end, which we describe in the next section. We describe now the data we used to train the system and the classes that are defined.

The success of our approach relies heavily on having available a reasonably large population of PDE problems whose characteristics span most of the space of all characteristic vectors. For the class of linear second order elliptic PDEs, PYTHIA uses the population defined in [9]. It consists of fifty–six linear, two–dimensional elliptic PDEs defined on rectangular domains. Forty–two of the problems are parameterized which leads to an actual problem space of more than two–hundred and fifty problems. Many of the PDEs were artificially created so as to exhibit various mathematical behaviors of interest; the others are taken from "real world" problems in various ways. The population has been structured by introducing measures of complexity of the operator, boundary conditions, solution and problem. The database created using this problem population contains information about the properties of the problems plus performance data for about 15,000 computations to solve one of the PDEs.

From this population, we define the following non–exclusive classes (the number of problems belonging to a class is given in parantheses):

1. SOLUTION–SINGULAR: Problems whose solutions have at least one singularity (6).

2. SOLUTION–ANALYTIC: Problems whose solutions are analytic (35).

3. SOLUTION–OSCILLATORY: Problems whose solutions oscillate (34).

4. SOLUTION–BOUNDARY–LAYER: PDE Problems with a boundary layer in their solutions (32).

5. BOUNDARY–CONDITIONS–MIXED: Problems that have mixed boundary conditions (74).

3 Methods of Class Selection

We have used several different methods, neural as well as non neural to identify the class(es) to which a new problem belongs. In this section, we describe the methods that were used, omitting details in the cause of brevity.

3.1 Traditional method

PYTHIA was originally implemented using a naive heuristic, which represented a problem class as the centroid of all the known exemplars of the class. The characteristic vector for a problem class was the average, computed element–by–element, of the characteristic vectors of all the class members. That is, the i^{th} element of the characteristic vector $CV(\cdot)$ of a class C is computed as:

$$(CV(C))_i = \frac{1}{|C|} \sum_{p \in C} (CV(P))_i.$$

where $|C|$ denotes the number of instances of PDEs in class C. The distance from a problem p to a class C is defined as the norm of the difference of the two characteristic vectors:

$$d(p, C) = \|CV(p) - CV(C)\|.$$

The norm can be chosen as any reasonable distance measure. Then, we say that p belongs to class C if $d(p, C) < \epsilon$ where ϵ is some threshold value that can be adjusted depending on the reliability of the characteristic vectors.

3.2 Feed Forward Neural-Nets : Gradient Descent Algorithms

Let us view the class selection problem as a mapping problem and suppose that we represent the m classes by a vector of size m. Suppose a 1 in the i^{th} position of the vector indicates membership in the i^{th} class. Our problem now becomes one of mapping the characteristic vector of size n into the classification vector of size m.

Feed forward neural networks have been shown to be effective in this task. We summarize here the various training techniques used.

We can use a backpropagation based neural network to determine this mapping. This network is essentially a supervised learning system consisting of an input layer, an output layer and one or more hidden layers, each layer consisting of a number of neurons. Each neuron has a state (s_i), and each connection between the i^{th} and j^{th} neuron has a weight w_{ij}. The neuron's output o_i uses some squashing function, f, applied to the weighted sum of all the inputs to the neuron. Mathematically

$$s_i = \sum_j w_{ij} o_j$$

$$o_i = f(s_i) = \frac{1}{1 + e^{-s_i}}$$

Using the backpropagation algorithm, the weights are then changed in a way so as to reduce the difference between the desired and actual outputs of the neural network. The weight changes Δw_{ij} are given by :

$$\Delta w_{ij} = \eta \delta_j o_i$$

where

$$\delta_j = \begin{cases} f_j'(net_j)(t_j - o_j), & \text{if unit } j \text{ is a output neuron;} \\ f_j'(net_j)(\sum_k \delta_k w_{jk}), & \text{if unit } j \text{ is a hidden neuron.} \end{cases}$$

where t_j is the teaching input of unit j and net_j is the net input to unit j, f' denotes the derivative of f and η is the "learning rate". This is essentially using gradient descent on the error surface with respect to the weight values. For more details, see the classic text by Rumelhart & McClelland [10]. Since the input and output of the network are fixed by the problem, the only layer whose size had to be determined is the hidden layer. We arbitrarily chose this to have 10 elements. Also, since we had no *a priori* information on how the various input characteristics affect the classification, we chose not to impose any structure on the connection patterns in the network. Our network was thus *fully connected*, that is, each element in one layer is connected to each element in the next layer. Thus there are only $32 \times 10 + 10 \times 5 = 370$ connections in the network, a relatively small number.

The second algorithm we consider modifies backpropagation by adding a fraction (the momentum parameter, α) of the previous weight change during the computation of the new weight change[2]. This simple artifice helps moderate changes in the search direction, reduce the notorious oscillation problems common with gradient descent. To take care of the "plateaus", a "flat spot elimination constant" λ is added to the derivative of f. Typical values of the momentum parameter are $(0 \ldots 1)$ and the flat spot elimination constant λ takes values from 0 to 0.25. The net effect of these enhancements is that (a) flat spots of the error surface are traversed relatively fast with few big steps, (b) the step size is decreased as the surface gets rougher, and (c) the search direction changes more slowly. This increases the learning speed significantly.

Quickpropagation (QuickProp)[3], uses information about the curvature (and second derivative) of the error surface to compute the weight change. QuickProp approximates the error surface to be locally quadratic and attempts to jump in one step from the current position directly into the minimum of the quadratic. This helps take care of "ridges" in the error surface. The important parameters here are μ, the maximum growth parameter, and ν, the weight decay term, μ is the maximum amount of the weight change that is added to the current change and ν, the weight decay term, is a factor to shrink the weights. ν is added to the slope S computed for each weight. This keeps the weights within an acceptable range and prevents problems like floating point overflow errors during computations. Values of μ are usually $1.75 \ldots 2.25$ and ν typically assumes low values like 0.0001 because QuickProp is very sensitive to it.

The final algorithm that we consider is called "Resilient backpropagation" (RProp)[1] because it uses the local topology of the error surface to make a more appropriate weight change. In other words, we introduce a 'personal update value' for each weight, which evolves during the learning process according to its local view of the error function. Thus we have two sets of learning equations, one for the weights and one for the update values themselves. RProp is very powerful and efficient beacuse the size of the weight step taken is no longer influenced by the size of the partial derivative. It is uniquely determined by the sequence of the signs of the derivatives, which provides a reliable hint about the topology of the local error function. At the beginning of the training, all the update values are set to an initial value, say Δ_0. As it is adapted as learning proceeds, the choice of Δ_0 is not critical. However, we set an upper bound on the update values Δ_{max}, so that learning avoids any unreasonably high values of weight steps. We set a default value of 0.1 to Δ_0 and Δ_{max} was varied in the range $0.1 \ldots 25$.

3.3 LVQ

LVQ (Learning Vector Quantization) borrows ideas from classical clustering and vector quantization techniques for signal processing, such as the k-nearest neighbor algorithm. Signal values are approximated by quantized references or 'codebook' vectors m_i. Several 'codebook' vectors are assigned to each class in the domain, and a new pattern x is said to belong to the same class to which the nearest m_i belongs. LVQ determines effective values for the 'codebook' vectors so that they define the optimal decision boundaries between classes, in the sense of bayesian decision theory. The accuracy and time needed for learning depend on an appropriately chosen set of codebook vectors and the exact algorithm that modifies the codebook vectors. We detail four different implementations of the LVQ algorithm - LVQ1, OLVQ1, LVQ2 and LVQ3. LVQ_PAK [7], a LVQ program training package was used in the simulation.

Let x be the input to the LVQ program and let m_c denote the 'codebook' vector closest to x. Then, the

```
main()
{
    for each set of labeled data i from training set do
        for each class j that pattern i belongs to
            box = identifyexpandablebox();
            if (box = NOTAVAILABLE) addnewbox();
            else expandbox(box);
            flag = checkforoverlap();
            if (flag = true) contracthyperboxes();
}
```

Figure 2: The neuro-fuzzy classification algorithm

'codebook' vectors are updated according to the simple rules :

$$m_c(t+1) = \begin{cases} m_c(t) + \alpha(t)[x - m_c(t)], & \text{if } x \text{ and } m_c \text{ belong to the same class;} \\ m_c(t) - \alpha(t)[x - m_c(t)], & \text{if } x \text{ and } m_c \text{ belong to different classes} \end{cases}$$

For all other codebook vectors, $m_i(t+1) = m_i(t)$. The control parameter α is not constant but varying with time. Normally, a linear decrease in time from a value of, say 0.1, is used.

OLVQ1 (Optimized LVQ1) is a modification of LVQ1 in which each 'codebook' vector m_i has its own learning rate α_i. It has been shown that the optimal value of α can be recursively defined as

$$\alpha_c(t+1) = \begin{cases} \frac{\alpha_c(t)}{1+\alpha_c(t)}, & \text{when } x \text{ is classified correctly;} \\ \frac{\alpha_c(t)}{1-\alpha_c(t)}, & \text{when } x \text{ is incorrectly classified.} \end{cases}$$

In practice, $\alpha_c(t)$ is allowed to increase steadily, but not above 1. Also, in LVQ_PAK, $\alpha_c(t)$ is never allowed to rise higher than its initial value.

The classification procedure in LVQ2 is similar to LVQ1, except that two 'codebook' vectors, m_i and m_j that are the nearest neighbors to x are now updated simulatneously. One of them is chosen to belong to the correct class and the other to a 'wrong' class. Also, these two vectors are selected so that x falls into a 'window' of values defined around the midplane of m_i and m_j. Thus LVQ2 *differentially* shifts the decision borders towards the bayes limits. Then the equations for updating the 'codebook' vectors become :

$$m_i(t+1) = m_i(t) - \alpha(t)[x - m_i(t)]$$

$$m_j(t+1) = m_j(t) + \alpha(t)[x - m_j(t)]$$

where m_i and m_j are the two closest 'codebook' vectors, x and m_j belong to the same class, and m_i and x belong to different classes.

It can be argued that LVQ2 might update the 'wrong' class vectors m_i too much so that the m_i vectors do not perform a good job of approximating the class distributions. LVQ3 introduces corrections that take care of this problem. In addition to the conditions mentioned for LVQ2, x should fall into the window defined by the vectors

$$m_k(t) + \epsilon\alpha(t)[x - m_k(t)] \text{ and}$$

$$m_l(t) + c\alpha(t)[x - m_l(t)]$$

where m_k, m_l and x belong to the same class. Typical values of ϵ are $0.1 \ldots 0.5$. The optimal value of c is found to decrease as the window size increases.

3.4 Fuzzy Neural Networks

We have developed a new neuro-fuzzy classification scheme suited for this problem. It is based on an algorithm proposed by Simpson[11]. The basic idea is to use fuzzy sets to describe pattern classes. These fuzzy sets are, in turn, represented by the fuzzy union of several n-dimensional hyperboxes. Such hyperboxes

define a region in n-dimensional pattern space that contain patterns with full-class membership. A hyperbox is completely defined by its min-point and max-point and also has associated with it a fuzzy membership function (with respect to these min-max points). This membership function helps to view the hyperbox as a fuzzy set and such "hyperbox fuzzy sets" can be aggregated to form a single fuzzy set class. This provides degree–of–membership information that can be used in decision making. The resulting structure fits neatly into a neural network assembly. Learning in the fuzzy min-max network proceeds by placing & adjusting the hyperboxes in pattern space. Recall in the network consists of calculating the fuzzy union of the membership function values produced from each of the fuzzy set hyperboxes.

Simpson's method assumes that the pattern classes underlying the domain are mutually exclusive and that each pattern belongs to exactly one class. But the pattern classes that characterize problems in many real world domains are frequently *not* mutually exclusive. For example, consider the problem of classifying geometric figures into classes such as polygon, square, rectangle etc., Note that these classes are not mutually exclusive (i.e., a square should be classified as a square *and* a rectangle *and* a polygon). It is possible to apply simpson's algorithm to this problem by first 'reorganizing' the data into mutually disjoint classes such as 'rectangles that are not squares', 'polygons that are not rectangles', and 'polygons' etc., but this strategy does not reflect the natural overlapping characteristics of the underlying base classes. In the PYTHIA domain, PDEs are classified on the basis of the mathematical properties possessed by their numerical solutions. Again, some PDEs might have an 'analytic solution', some might have 'mixed boundary conditions', but some PDEs can both be 'analytic' *and* have 'mixed boundary conditions'.

Thus Simpson's algorithm fails to account for a situation where one pattern might belong to several classes. Also, the only parameter in the Simpson's method is the maximum hyperbox size parameter θ (The sensitivity parameter γ is normally set to a constant so as to produce a moderately quick decrease from full membership to no membership). It is not reasonable to assume that one parameter is sufficient to tune the entire system. Moreover, as mentioned in [11], the effect of θ on classification accuracy is not completely understood.

In this section, we develop an enhanced scheme that operates with such overlapping *and* non-exclusive classes. In this process, we introduce another parameter δ to tune the system. We then study the effect of the parameters θ and δ on classification accuracy by applying the method to a real-world problem in scientific computation.

Consider the k^{th} ordered pair $\{A_k, d_k\}$ from the training set. Let the desired output for the k^{th} pattern be $[1, 1, 0, 0, ..., 0]$. The algorithm in fig. 2 considers this as two ordered pairs containing the same pattern A_k but with two pattern classes as training outputs - $d_{k1} = [1, 0, 0, 0, ..., 0]$ and $d_{k2} = [0, 1, 0, 0, ..., 0]$ respectively. In other words, the pattern is associated with both class 1 and class 2. This will cause hyperboxes of both classes 1 and 2 to completely contain the pattern A_k. But according to Simpson's original algorithm, one ordered pair can have complete membership in only hyperboxes of the same class. In other words, the algorithm in Fig. 2 will perceive this as undesirable overlap and contract the hyperboxes. It will be seen in the course of this section that the contraction step will cause the pattern to have complete membership in neither of the classes. Thus, the above procedure results in the pattern having equal degrees of membership in both the hyperboxes but is not completely contained in either of them.

Assume that the network is first trained with the desired output as $d_{k1} = [1, 0, 0, 0, ..., 0]$. This results in the k^{th} pattern A_k having complete containment in a hyperbox of class 1 (because the $1st$ bit is set to 1). Then when we train the same pattern with $[0, 1, 0, 0, ..., 0]$, a hyperbox of class 2 will be created/expanded to include the k^{th} pattern. This will result in hyperbox overlap. The hyperbox contraction step detailed below ensures that both the hyperboxes are adjusted so that each of them contain the k^{th} pattern to the same degree (which will be less than 1).

(a) **Hyperbox Expansion** : Given labeled data of the form $\{A_h, d_h\}$, find the hyperbox b_j that represents the class d_h, provides the highest degree-of-membership and allows expansion (if needed). Since we bound the maximum hyperbox size by θ, the following condition is satisfied :

$$\frac{1}{n}\sum_{i=1}^{n}(max(w_{ji}, a_{hi}) - min(v_{ji}, a_{hi})) \leq \theta \tag{6}$$

Then, the min-points and the max-points are adjusted by the equations :

$$v_{ji}^{(k+1)} = min(v_{ji}^{(k)}, a_{hi}) \; \forall i = 1, 2, \ldots, n \tag{7a}$$

$$w_{ji}^{(k+1)} = max(w_{ji}^{(k)}, a_{hi}) \; \forall i = 1, 2, \ldots, n \tag{7b}$$

(b) **Overlap Testing** : A dimension-by-dimension comparison between hyperboxes is effected here. This test is conducted between the hyperbox expanded in the previous step and any other hyperbox that represents a different class. Let B_j be the one expanded in the previous step and B_l represent another hyperbox of a different class. If at least one of the following conditions is satisfied for a dimension, then we conclude that overlap exists between the hyperboxes. $\Delta^{(k)}$ is initialized to 1. Figures 3 to 8 indicate a two-dimensional case where overlap has been detected along the first dimension i.e., along the abscissa. The various conditions to be tested for are as follows :

Condition 1 (Fig. 3) : $v_{ji} < v_{li} < w_{ji} < w_{li}$

$$\Delta^{(k+1)} = min(w_{ji} - v_{li}, \Delta^{(k)}) \tag{8a}$$

Condition 2 (Fig. 4) : $v_{li} < v_{ji} < w_{li} < w_{ji}$

$$\Delta^{(k+1)} = min(w_{li} - v_{ji}, \Delta^{(k)}) \tag{8b}$$

Condition 3 (Figs. 5 and 6) : $v_{ji} < v_{li} < w_{li} < w_{ji}$

$$\Delta^{(k+1)} = min(w_{li} - v_{ji}, w_{ji} - v_{li}, \Delta^{(k)}) \tag{8c}$$

Condition 4 (Figs. 7 and 8) : $v_{li} < v_{ji} < w_{ji} < w_{li}$

$$\Delta^{(k+1)} = min(w_{ji} - v_{li}, w_{li} - v_{ji}, \Delta^{(k)}) \tag{8d}$$

If $\Delta^{(k+1)} > \Delta^{(k)}$, then there was no overlap and the next contraction step is unnecessary. If, on the other hand, $\Delta^{(k+1)} < \Delta^{(k)}$, then overlap has occured in the i^{th} dimension and the $(i+1)^{th}$ dimension is now checked for overlap after setting $\Delta^{(k)} = \Delta^{(k+1)}$.

(c) **Hyperbox Contraction** : If overlap was detected in the i^{th} dimension, as detailed above, we minimally adjust the i^{th} dimensions of each of the overlapping hyperboxes. In other words, we try to adjust the hyperboxes so that only one of the min/max points is altered at a time. We examine the same four cases as above,

Condition 1 (Fig. 3) : $v_{ji} < v_{li} < w_{ji} < w_{li}$

$$w_{ji}^{(k+1)} = v_{li}^{(k+1)} = \frac{w_{ji}^{(k)} + v_{li}^{(k)}}{2} \tag{9a}$$

Condition 2 (Fig. 4) : $v_{li} < v_{ji} < w_{li} < w_{ji}$

$$w_{li}^{(k+1)} = v_{ji}^{(k+1)} = \frac{w_{li}^{(k)} + v_{ji}^{(k)}}{2} \tag{9b}$$

Condition 3a (Fig. 5) : $v_{ji} < v_{li} < w_{li} < w_{ji}$ and $(w_{li} - v_{ji}) < (w_{ji} - v_{li})$

$$v_{ji}^{(k+1)} = w_{li}^{(k)} \tag{9c1}$$

Condition 3b (Fig. 6) : $v_{ji} < v_{li} < w_{li} < w_{ji}$ and $(w_{li} - v_{ji}) > (w_{ji} - v_{li})$

$$w_{ji}^{(k+1)} = v_{li}^{(k)} \tag{9c2}$$

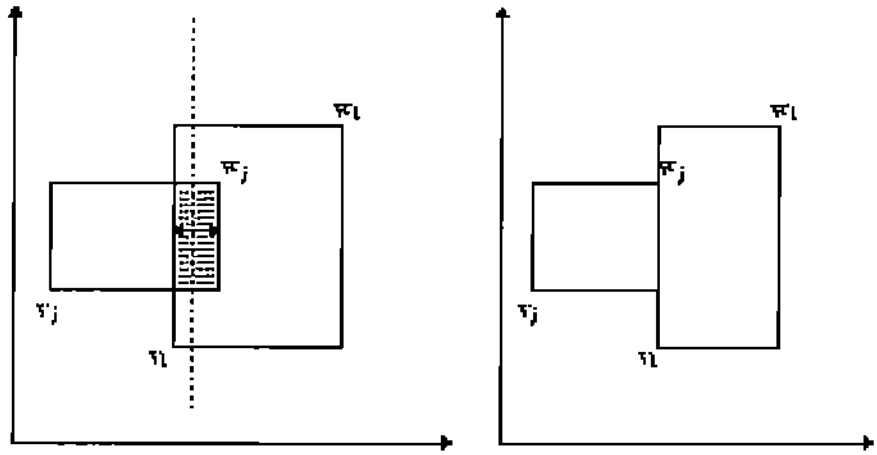

Figure 3: Condition 1 : $v_{ji} < v_{li} < w_{ji} < w_{li}$

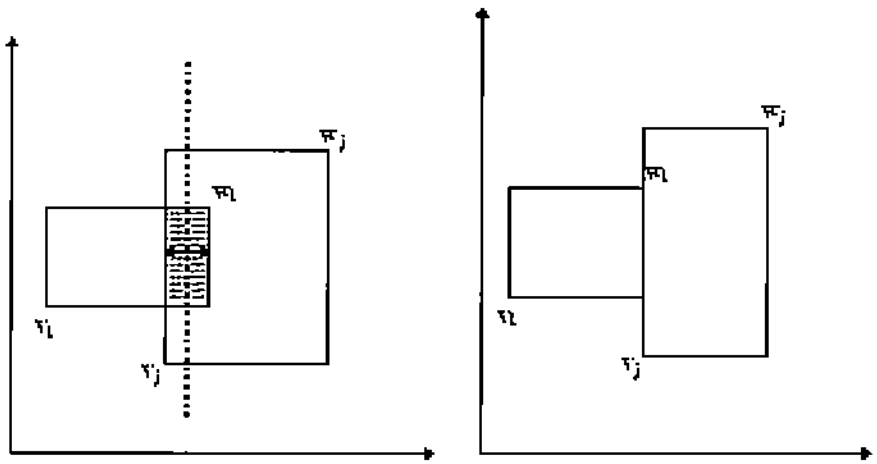

Figure 4: Condition 2 : $v_{li} < v_{ji} < w_{li} < w_{ji}$

Condition 4a (Fig. 7) : $v_{li} < v_{ji} < w_{ji} < w_{li}$ and $(w_{li} - v_{ji}) < (w_{ji} - v_{li})$

$$w_{li}^{(k+1)} = v_{ji}^{(k)} \tag{9d1}$$

Condition 4b (Fig. 8) : $v_{li} < v_{ji} < w_{ji} < w_{li}$ and $(w_{li} - v_{ji}) > (w_{ji} - v_{li})$

$$v_{li}^{(k+1)} = w_{ji}^{(k)} \tag{9d2}$$

Since each pattern can belong to more than one class, we need to define a new way to interpret the output of the fuzzy min-max neural network. In the original algorithm, we locate the node in the output layer with the highest value and set the corresponding bit to 1. All other bits are set to zero. In this way, a hard decision is obtained.

In the modified algorithm, however, we introduce a parameter δ and we set to 1 *not only* the node with the highest output *but also* the nodes whose outputs fall within a band $\pm\delta$ of the output value. This results in more than one output node getting included and consequently, aids in the determination of non-exclusive classes. It also allows us to include 'nearby classes' in our decision : Consider the scenario (Fig. 9) when a pattern (x in the figure) gets associated with the wrong class, say Class 1, merely because of its proximity to members of Class 1 that were in the training samples rather than to members of its characteristic class (Class 2). Such a situation can be caused due to a larger incidence of the Class 1 patterns in the training

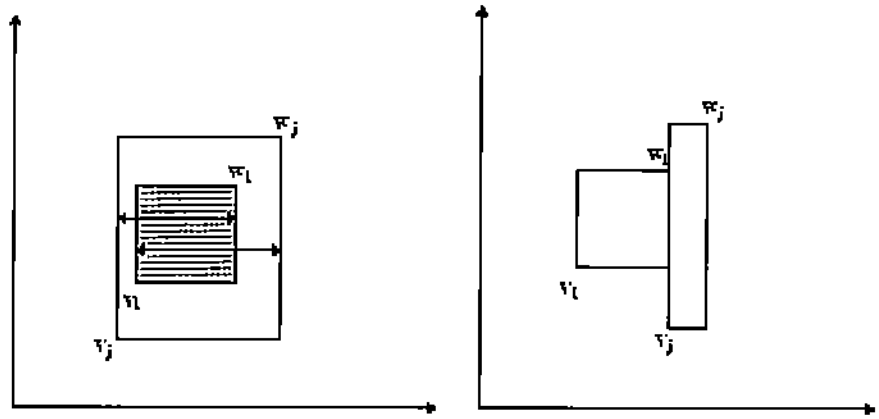

Figure 5: Condition 3a : $v_{ji} < v_{li} < w_{li} < w_{ji}$ and $(w_{li} - v_{ji}) < (w_{ji} - v_{li})$

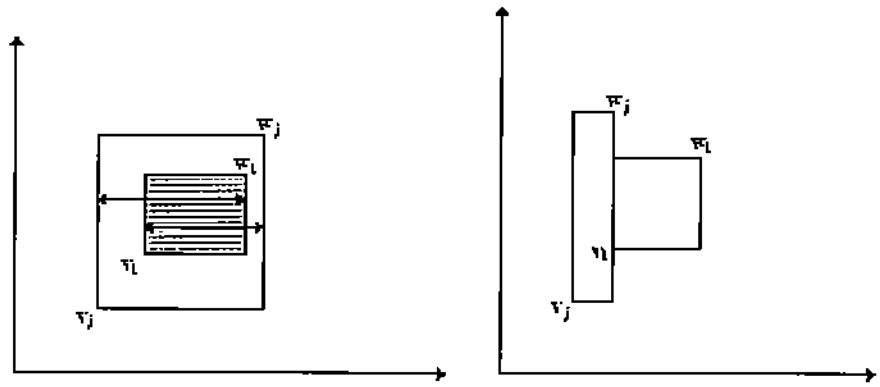

Figure 6: Condition 3b : $v_{ji} < v_{li} < w_{li} < w_{ji}$ and $(w_{li} - v_{ji}) > (w_{ji} - v_{li})$

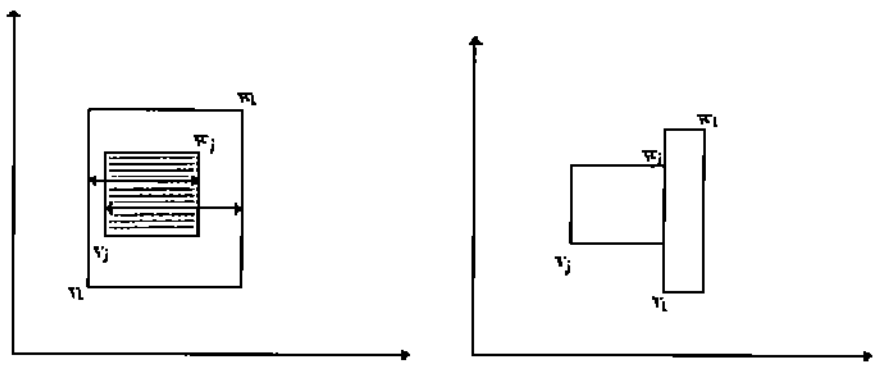

Figure 7: Condition 4a : $v_{li} < v_{ji} < w_{ji} < w_{li}$ and $(w_{li} - v_{ji}) < (w_{ji} - v_{li})$

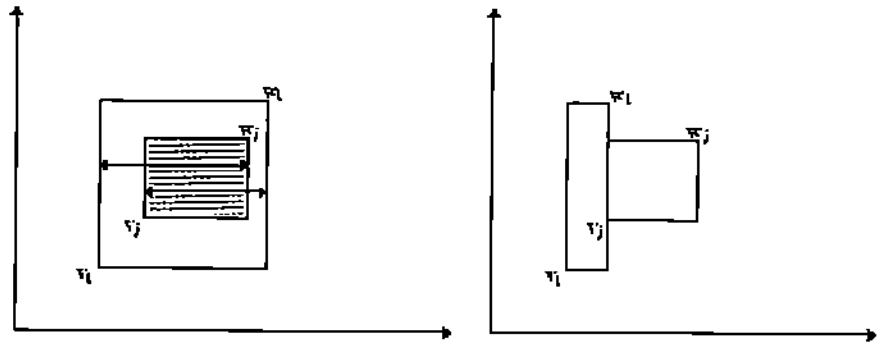

Figure 8: Condition 4b : $v_{li} < v_{ji} < w_{ji} < w_{li}$ and $(w_{li} - v_{ji}) > (w_{ji} - v_{li})$

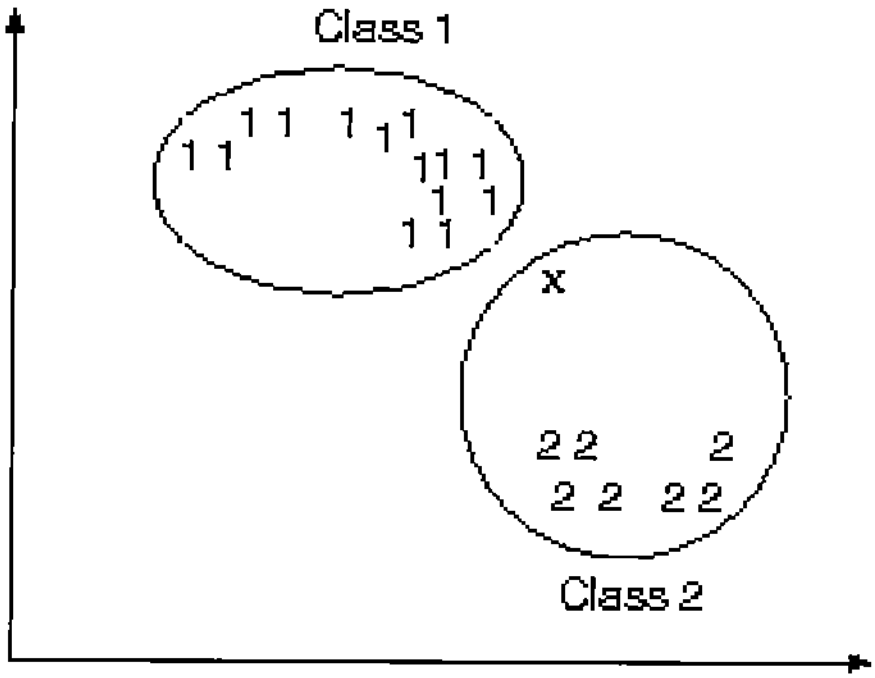

Figure 9: Recognizing "nearby" classes

set than the Class 2 patterns or due to a non uniform sampling, since we make no prior assumption on the sampling distribution. In such a case, the δ parameter gives us the ability to make a soft decision by which we can associate a pattern with more than one class.

4 Results from classification

In this section, we describe results from classification experiments we did on the PYTHIA domain using the techniques described earlier. In performing these experiments, a separate data set is used for "training" (i.e., in the modeling stage) and another separate data set is used to measure the "learning" and "generalization" provided by the paradigm (this is called the testing data set). The PYTHIA data set consists of a set of 167 ordered pairs $\{A_h, d_h\}$, where $A_h = (a_{h1}, a_{h2}, ..., a_{h32}) \in I^{32}$ is the input pattern (the 32-vector encoding of the PDE problem) and $d_h \in \{1, 2, ..., 5\}$ is the index of one of the 5 classes. The PYTHIA data set is split into two parts - the first part contained approximately 2/3 of the total exemplars, i.e., 111. The second part represents the other one-third of the PDE population i.e., 56 exemplars. Each paradigm described in the previous section was trained using both (i) the first part and the (ii) the second part. For this reason, we refer to (i) as the larger training set and (ii) as the smaller training set. After training, the learning

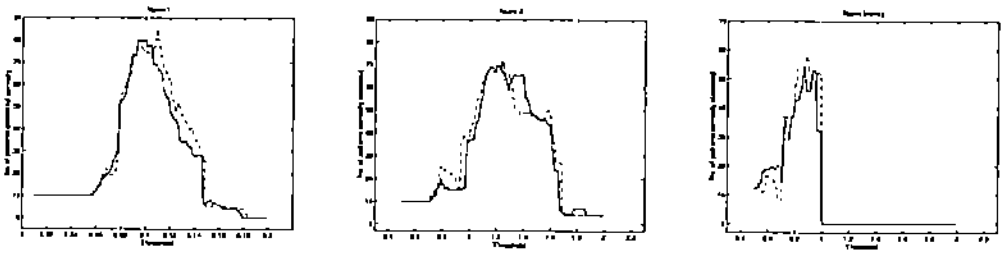

Figure 10: Performance of the traditional method using Norms L_1, L_2 and L_∞. The solid lines indicate the performance using the larger training set and the dashed lines indicate the performance using the smaller training set.

of the paradigm was tested by applying it to the entire PYTHIA data set of 167 exemplars. Each method previously discussed is operated with a wide range of the parameters that control its operation. For example, the performance of a simple backpropagation neural network was studied by varying its learning rate. We report the results from only the "best" set of parameters.

In each of these techniques, the number of patterns classified correctly was determined as follows : We fix a threshold for the L_2 error norm and infer that error vectors with corresponding error norms above the threshold have been incorrectly classified (the error vector is defined as the component-by-component difference between the desired output and the actual output). We have carried out experiments using threshold values of 0.2, 0.1, 0.05 and 0.005 for each of the techniques. Also, for each technique we concentrate on the key aspects of the paradigm only for the sake of conciseness.

However, the classes defined in PYTHIA are *not* mutually exclusive as described earlier. Of the methods discussed in the previous section, only feed forward neural networks inherently cater to mutually non-exclusive classes. The other paradigms require the user to associate a single class with the problem characteristic vector at the time of training Hence, in these paradigms, we interpret the data sets as follows : Consider the k^{th} ordered pair $\{A_k, d_k\}$ from the training set. Let the desired output for the k^{th} pattern be $[1, 1, 0, 0, ..., 0]$. We consider this as two ordered pairs containing the same pattern A_k but with two pattern classes as training outputs - $d_{k1} = [1, 0, 0, 0, ..., 0]$ and $d_{k2} = [0, 1, 0, 0, ..., 0]$ respectively. In other words, the pattern is associated with both class 1 and class 2.

4.1 Traditional method

It was shown in the previous section that the traditional method relies on the definition of an appropriate distance measure to quantify the distance of a problem P from a class C. We have used three definitions of the norm $\|\cdot\|$, namely the norms $\|\cdot\|_1$, $\|\cdot\|_2$ and $\|\cdot\|_\infty$. These norms can be defined as follows,

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$$

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

$$\|\mathbf{x}\|_\infty = max(|x_i|)$$

In the above equations $\mathbf{x}$ refers to a vector of n elements and x_i refers to the i^{th} element of the vector. Each of these norms was used in conjunction with both the larger training set and the smaller training set. The threshold value ϵ was varied within an appropriate range - i.e., each norm has a different interpretation of the distance and hence a single range of ϵ was not suitable for representing different metrics. For the norm $\|\cdot\|_1$, the range was $(0.01, 0.2)$. For the norms $\|\cdot\|_2$ and $\|\cdot\|_\infty$, the range was $(0.5, 2)$. Fig. 10 displays the number of patterns classified correctly as a function of ϵ for each of the above three norms. It was observed that varying the threshold, contrary to expectations, did not lead to a perceptible improvement/decline in the performance of the paradigm. It can be observed that $\|\cdot\|_1$ provides the best performance over the other two norms though the absolute accuracy provided by it seldom rises above 50%. Also training with

the smaller set, surprisingly leads to a better performance than training with the larger set (This can be seen from Fig. 10). This can be attributed to the possibility that the smaller set was more representative of the data in terms of the structure that the traditional method imposes on the distribution of the patterns. The larger training set, on the other hand, would have offered more inconsistencies to cope up with rather than providing "good" information for learning.

4.2 Feed Forward Neural Networks

As shown in the previous section, feed forward networks perform a mapping from the problem characteristic vector to an output vector describing class memberships. We have trained a 32x10x5 feed forward neural network using the algorithms studied in the previous section. Each of these networks was trained with five choices of the control parameters and the choice leading to the best performance was considered for performance evaluation. In other words, each network was trained to 2000 iterations, and at the end of this, its "learning" was evaluated. Again, as mentioned in the previous subsection, both the larger training set and the smaller set were used to separately train the network. Below we detail how each paradigm was evaluated. All the simulations were performed using the Stuttgart Neural Network Simulator[2].

As the only "free" parameter in the simple backpropagation paradigm was the learning rate η, it was varied in the range $[0.1 \ldots 0.9]$. It was observed that the best performance was achieved at a value of $\eta = 0.9$. In other words, increasing η led to an increase in convergence time, so that at the end of 2000 iterations, the number of patterns classified correctly was substantially higher than that for any other value of η.

In the variant of backpropagation we introduce a momentum term to speed up the convergence process and also to take care of phenomena like "local minima". Also the flat spot elimination constant ensures that the algorithm continues to traverse along reasonably flat portions of the search space. The important parameters here are the learning rate η, the momentum coefficient α and the flat spot elimination constant λ. η was kept at a low value (0.2), because of the overpowering effect of the high momentum term which was found to be "optimal" at the values $0.7, 0.8, 0.9$. The ideal values of the flat spot elimination constant was found to be 0.05 or 0.1. It was observed that for a flat spot elimination constant of 0.1, the network ran into lots of local minima problems and the weights got adjusted to very high values. For this reason, the best performance was achieved at $(\eta, \alpha, \lambda) = (0.2, 0.8, 0.05)$.

QuickProp also assumed a low value of the learning rate η. Also, the parameters μ, the maximum growth parameter and ν, the weight decay term influence the performance of QuickProp very much. It was observed that the ideal value of μ was in the range $[1.75 \ldots 2]$ and that for ν was either 0.0001 or 0.0002. QuickProp had a very fast convergence rate; even though it got into lots of local minima problems, it was always able to come out of them with very high momentum. Also, the maximum weight changes took place in the first $100 - 200$ iterations and the subsequent iterations only served to "fine-tune" the error attained in these initial iterations. The best performance was achieved at a value of 0.2 for η, 1.75 for μ and 0.0001 for ν.

Of all the supervised paradigms for feed forward neural networks studied in this article, RProp provided the maximum performance for the same number of training iterations. We chose a fixed value of Δ_0 because the algorithm refines it iteratively and we set an upper bound on the weight changes Δ_{max} of 25. Even though some local minima problems were observed at high values of Δ_{max}, an extremely fast convergence rate served to make the network settle to a comfortable error level in about 100 iterations. The best performance was achieved at $(\Delta_0, \Delta_{max}) = (0.1, 25)$.

Fig. 11 describes the behavior of the four methods for specific values of the L_2 error norm threshold. It can be observed from this figure that as the threshold value is decreased, the performance of backpropagation, enhanced backpropagation and QuickProp methods decline while that of RProp consistently maintains a high value. RProp manages to correctly classify 160 of the 167 patterns. The accuracy of backpropagation, enhanced backpropagation, QuickProp and RProp are for different values of the L_2 error norm are given as follows :

1. threshold = 0.005 : (47.3%, 72.45%, 74.25% and 95.83%)
2. threshold = 0.05 : (90.41%, 93.41%, 94.61% and 95.83%)
3. threshold = 0.1 : (92.81%, 94.01%, 94.61% and 95.83%)
4. threshold = 0.2 : (92.81%, 94.01%, 94.61% and 95.83%)

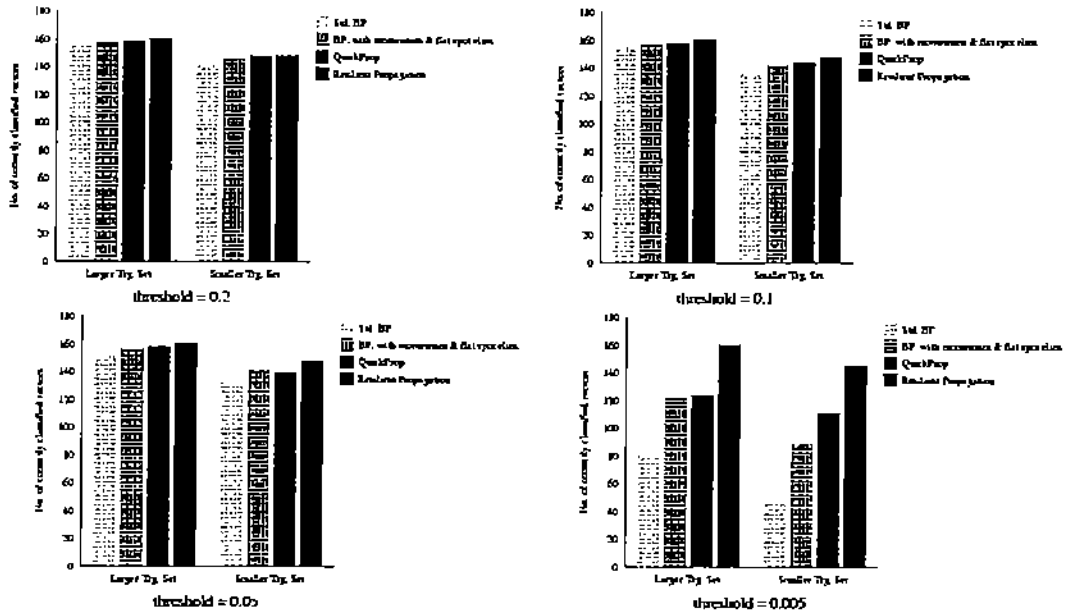

Figure 11: The performance of four feed forward networks for four threshold values

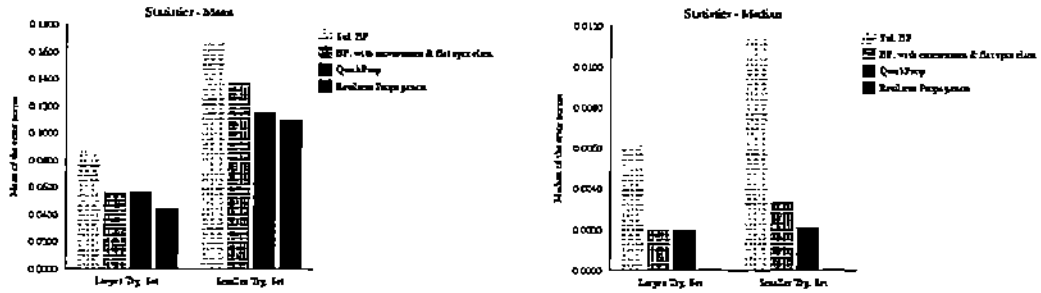

Figure 12: Mean and median values of the four feed forward neural networks

Fig. 12 gives another measure describing the situation. We present the mean and median values for the error norms in these graphs (for an L_2 error norm threshold of 0.005). Again, it can be observed from these graphs that RProp provides the best performance of all the feed forward neural network paradigms. It can be seen from Fig. 12 that RProp's median error is nearly negligible. Thus, while the mean value describes the average error, the very low median value of RProp shows us that while there are outliers, RProp classifies most of the problem patterns correctly. Also it is observed that using the smaller training set instead of the larger training set does lead to a degradation of performance.

4.3 LVQ

Since the LVQ algorithms and the fuzzy min-max neural network work for labeled data of the form $\{A_h, d_h\}$, there is an implicit assumption that each pattern should belong to at least one class. However, in the problem domain, we may come across instances of PDEs that do not belong to any of the above defined classes. To circumvent this difficulty, we define a sixth class as follows :

(vi) SPECIAL : PDE Problems whose solutions do not fall into any of the classes (i)–(v). This artifice is employed in the LVQ algorithms and the fuzzy min-max neural network described in the next section.
The LVQ algorithms mentioned in the previous section were trained as follows - 50 codebook vectors were chosen so that their numbers in the respective classes were proprtional to their a priori probabilities. Then the algorithms were trained for 2000 iterations using both the larger and the smaller training sets. Due to space limitations, we present here only the results with the larger training set.

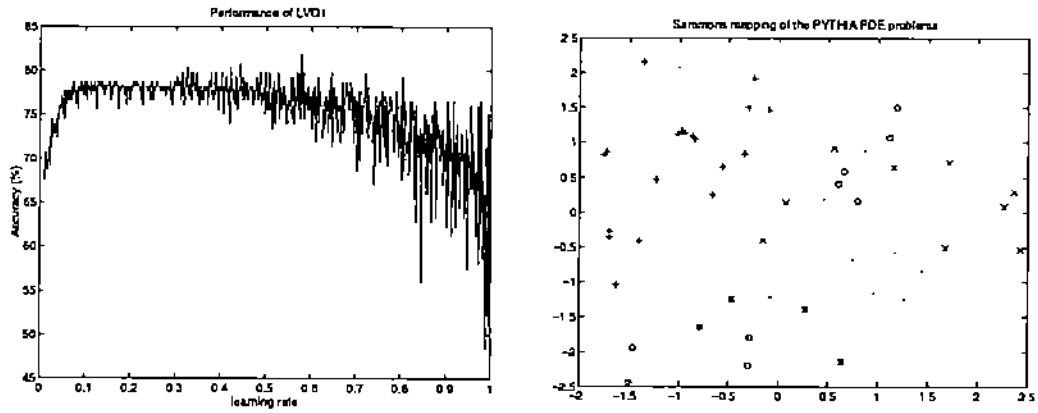

Figure 13: (a) Performance of the LVQ1 algorithm and (b) Clustering of the PDE problem classes

The important free parameter in LVQ1 was the learning rate. This was varied from 0.1 to 1 in steps of 0.01. The accuracy achieved is plotted against the learning rate in Fig. 13a.

The highest accuracy 77.06% was attained at a learning rate of 0.05 (this was for an L_2 threshold value of 0.005). LVQ1 is used to provide an "initial" solution and other LVQ algorithms can be used to improve the learning done by the LVQ1 algorithm. We adopt this strategy for our experiment.

OLVQ1 was subsequently trained for 200 iterations and the accuracy obtained was 80% (L_2 threshold value = 0.005). Thus OLVQ1 substantially fine-tunes the initial solution provided by LVQ1. In Fig. 13b, we map the 32-dimensional data space of the codebook vectors onto the two-dimensional plane. We use Sammon's mapping to achieve this. The two-dimensional mapping approximates to the euclidean distances of the data space and thus visualizes the clustering of data. In Fig. 13b, the dots represent the analytic PDEs, the 'o's denote the oscillatory PDEs, the 'x's denote the boundary layer PDEs, the '+' signs denote the boundary-conditions-mixed PDEs, the asterisks denote the singlar entities and the single entry denoted by a '-.' represents a PDE which does not belong to any of the above classes.

The LVQ2 algorithm depends on the window width parameter i.e., the relative 'width' of the window into which the training data must fall. We varied the window width parameter from 0.1 to 0.5 and also the learning rate as mentioned in the LVQ1 experiment. It was observed that the optimal performance was achieved at a window width of 0.3 and a learning rate of 0.2. However the accuracy for this implementation (for an L_2 error threshold of 0.005) was only 79.79%, a bit lower than that achieved by the OLVQ algorithm.

The LVQ3 algorithm can be used for an additional fine-tuning stage in learning. The relative learning rate parameter ϵ is used (multiplied by the parameter α), when both the nearest codebook vectors belong to the same class. Again, as in the LVQ3 experiment, the relative window width parameter determines the "box" into which the trainig data must fall. Again, a window size of 0.3 was used and the relative learning rate parameter ϵ was set at 0.1. It was observed that though LVQ3 improves the initial codebook, it does not guarantee results better than the OLVQ algorithm. For example, the maximum accuracy attained by the LVQ3 algorithm was 79.26% (for an L_2 error threshold value of 0.005).

4.4 Fuzzy Min-Max Neural Networks

The following set of experiments were conducted (again, we present the results only with the larger training set) :

(i) **Effect of θ** : In this set of experiments, the max. hyperbox size was varied continuously and its effect on other variables were studied. In particular, it is observed that when θ was increased, a lesser number of hyperboxes needed to be formed, i.e., when θ tends to 1, the number of hyperboxes formed is 6 - the number of classes in the domain. Also performance on the training set and the test set steadily improved as θ was decreased (Fig. 14). Performance on the training set was, expectedly, better than that on the test set. An optimal error was achieved at a θ value of 0.00125. When $\theta > 0.00125$, the error increased on both the sets and when $\theta < 0.00125$, the network *overfit* the training data so that its performance on the test set started to decline. The number of hyperboxes formed for this optimal value of θ was 62 - approximately double the

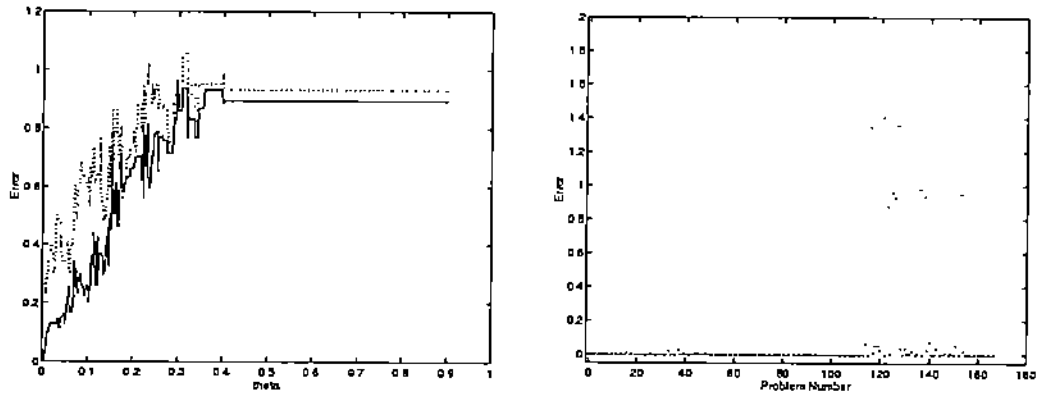

Figure 14: The graph on the left shows the effect of θ on the performance. The solid line indicates the error on the training set while the dashed line indiactes the error on the test set. The graph on the right is the scatter plot of results for optimum θ and $\delta = 0.01$.

size of the dimension of the pattern space.

(ii) **Effect of δ** : In this experiment, we set $\theta = 0.00125$ (the optimal value) and we vary the threshold δ by assigning to it the values 0.01, 0.02, 0.05 and 0.09. It is observed that when δ was increased, more output nodes tend to get included in the "reading-off" stage so that the overall error increased. Fig. 14 shows a scatter plot of the results for $\delta = 0.01$.

(iii) **On-line adaptation** : The last series of experiments conducted were to test the fuzzy min-max neural network for its on-line adaptation, i.e., each pattern was incrementally presented to the network and the error on both sets was recorded at each stage. It was observed that the number of hyperboxes formed slowly increases from 1 to the optimal number 62 in Expt.(i). Also, performance on both sets steadily improved to the values obtained in Expt.(i).

The accuracy obtained by the fuzzy min-max neural network is 95.21% for L_2 error norm thresholds of 0.005, 0.05, 0.1 and 0.2 (Varying the L_2 threshold value did not alter the accuracy).

4.5 Discussion

In the table below, we summarize the mean and median values for all the paradigms discussed in this article :

Method	Description	Mean	Median
Traditional	Norm 1	0.618657	1.00000
Feed Forward	RProp	0.044661	0.000001
LVQ	OLVQ1	0.155557	0.006125
Fuzzy MinMax	—	0.055345	0.000001

In each of the rows in the above table, the best possible method and the optimal combination of the parameters was used for the comparison. It was observed that the naive technique which represents classes by the centroid of the known samples performed very poorly (with an accuracy of 47.9%). Feed forward neural networks, in general, performed quite well, with more complicated training schemes such as enhanced backpropagation, Quick Propagation, Resilient Propagation clearly winning over plain error backpropagation. For higher L_2 error threshold values (say 0.2), all these learning techniques gave values close to each other (92.81%, 94.01%, 94.61% and 95.83% respectively). However, when the L_2 error threshold levels were lowered (to 0.005), RProp clearly won out on all the other methods (with an accurracy of 95.83% over 47.3%, 72.45% and 74.25% for plain backpropagation, enhanced backpropagation and quick propagation). The same observations can be made by looking at the mean and median of the error values. While the mean for RProp (0.0446) is slightly lower than that of others, the median is significantly lower (0.000001). This means that RProp classifies most patterns correctly with almost zero error, but has few outliers. The other methods have

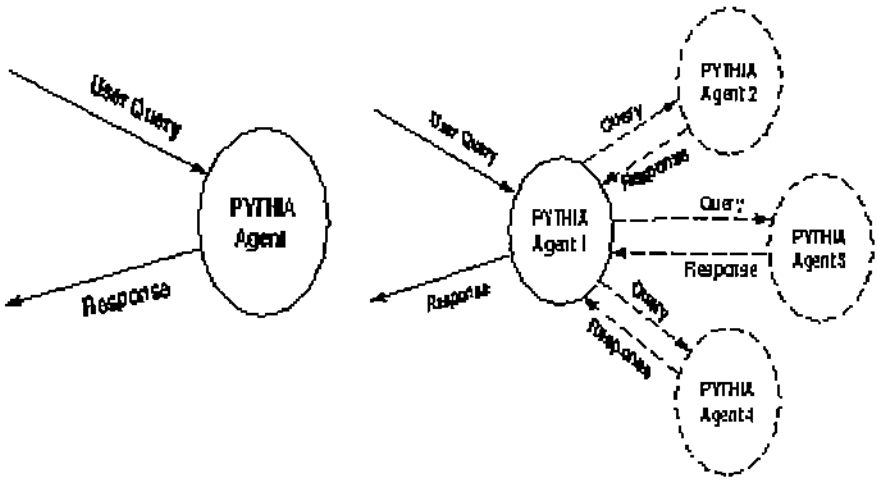

Figure 15: The PYTHIA Collaborative MultiAgent System

the errors spread more "evenly", which leads to a degradation in their performance as compared to RProp. The variants of the LVQ method (LVQ1, OLVQ1, LVQ2 and LVQ3) that we tried performed about average. While they were better than the naive classifier (with an accuracy of 80% for OLVQ1), their performance was only in the $75-80\%$ range (for an L_2 error threshold value of 0.005). Increasing the L_2 error threshold value did not serve to improve the accuracy. Finally, our neuro-fuzzy method, which is a variant of that proposed by Simpson[11] was observed to perform quite well. In fact, it performed almost as well as RProp both in terms of % accuracy (95.20%), mean error (0.05534) and median error (0.000001). Like RProp, increasing the L_2 error threshold did not significantly alter the performance. Considering that unlike RProp, our method allows on-line adaptation (i.e., new data don't require retraining on the old data), it is clearly superior in this context. This is because, in the PYTHIA environment, we expect the system to constantly update its database with the new problems it has seen.

5 Collaborative PYTHIA

Recently, we have begun to move towards making PYTHIA a collaborative multiagent system[6]. This is, as we shall illustrate, a more natural implementation. PDEs can be widely varying. Most application scientists tend to solve only a limited kind, and hence any PYTHIA agent they are running is likely to be able to answer questions effectively only about a limited range of problems. If there were mechanisms that allowed PYTHIA agents of various application scientists to collaborate, then each agent could share knowledge and potentially answer a broader range of questions – call upon the collective wisdom of all agents, as it were (Fig. 15).

Assuming that there is a multitude of PYTHIA agents available, the question then becomes one of trust. In other words, if different agents we ask give us different answers, which answer do we accept as correct ? Further, can we learn about the behaviour of other agents, and rather than asking everyone the question, we ask only those whose answers we are likely to trust, or for whom we know that a reasonable answer will be forthcoming. We have recently proposed a new coordination strategy for a multiagent PYTHIA setting which combines epsitemic utility and our neuro-fuzzy learning technique to form an unsupervised learning environment. The basic use of learning here is to obtain a mapping from problem characteristics to the PYTHIA agent which is likely to give the best method for its solution. We have made some progress in this direction, and a suitably trained fuzzy min max neural netowork maps the problem characteristics to one/more of several PYTHIA agents that can provide a solution for the problem.

6 Conclusion

In this paper, we have described Problem Solving Environments, which are a key factor in the development of High Performance Scientific Computing. We also described PYTHIA, a system that advises the user on appropriate selection of software and hardware systems and parameters for solving scientific computing problems. PYTHIA makes decisions in the presence of imprecise information, and needs the ability to generalize from previously seen exemplars to new data. An important component of PYTHIA is the classification component which identifies the class(es) to which a given problem belongs. We have used several traditional and neural/neuro-fuzzy techniques to implement this classification system. Results and comparative performance of these techniques are presented. It was observed that Resilient Propagation and the fuzzy min max neural network were the most promising of the paradigms considered in terms of accuracy of representation. Further work on the neural aspects of PYTHIA is in progress in several dimensions. We are developing a method to directly map the original problem, that of selecting a method for a problem provided the user's estimates of the required time/grid and error criterion, to a Neural Network. While this leads to an increased learning time, the decision process would be virtually instantaneous, especially if we exploit the SIMD parallelism inherent in the network. Also, we are investigating extensions to predict when one would need to use a parallel machine and when necessary, what machine to use and what its configuration should be. Work is also in progress on improving our neuro-fuzzy scheme to use non-isothetic hyperboxes (i.e., hyperboxes that do not necessarily have their sides aligned to the orthogonal axes) and hyperspheres to cover the pattern space.

References

[1] H. Braun and M. Riedmiller. Rprop : A Fast and Robust Backpropagation Learning Strategy. In *Proceedings of the ACNN*, 1993.

[2] Zell et.al. Snns : Stuttgart neural network simulator. Technical Report 3/93, Institute for Parallel and Distributed High Performance Systems, University of Stuutgart, Fed Rep. of Germany, 1993.

[3] S.E. Fahlman. Faster-learning Variations on Backpropagation : An Empirical Study. In T.J. Sejnowski, G.E. Hinton, and D.S. Touretzky, editors, *in 1988 Connectionist Models Summer School.* Morgan Kaufmann, 1988.

[4] E. Gallopoulos, E. Houstis, and J.R. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science and Enginerring,*, vol.1(2):pp.11–23, 1994.

[5] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, Ko-Yang Wang, and S. Weerawarana. //ELLPACK: A numerical simulation programming environment for parallel MIMD machines. In J. Sopka, editor, *Proceedings of Supercomputing '90*, pages 96–107. ACM Press, 1990.

[6] Anupam Joshi. To Learn or Not to Learn ... In *Proc. IJCAI'95 Workshop on Adaptation and Learning in Multiagent Systems*, 1995. (to appear).

[7] T. Kohonen, J. Kangas, J. Laaksoonen, and K. Torkolla. LVQ-PAK Learning Vector Quantization Program Package. Technical Report 2C, Laboratory of Computer and Information Science, Rakentajanaukio, 1992.

[8] D.P. O'Leary. Parallel Computing : Emerging from a Time Warp. *IEEE Computational Science and Enginerring,*, vol.1(3), 1994.

[9] John R. Rice, Elias N. Houstis, and Wayne R. Dyksen. A population of linear, second order, elliptic partial differential equations on rectangular domains, part I. *Mathematics of Computation*, 36:475–484, 1981.

[10] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Internal Representations by Error Propagation. In D.E. Rumelhart and McClelland J.L., editors, *in Parallel Distributed Processing : Explorations in the Microstructure of Cognition*, volume I. The MIT Press, 1986.

[11] P.K. Simpson. Fuzzy Min-Max Neural Networks–Part 1: Classification. *IEEE Transactions on Neural Networks*,, vol.3(5):pp.776–786, 1992.